AD-A115 738   NAVAL RESEARCH LAB WASHINGTON DC   F/G 17/2
AN APPROACH TO DESCRIBING THE FUNCTIONAL REQUIREMENTS OF AN EMB--ETC(U)
JUN 82 C L HEITMEYER, J D MCLEAN
UNCLASSIFIED NRL-8604   NL

1 OF 1
AD A
115738

END
DATE
FILMED
7-82
DTIC

2

NRL Report 8604

An Approach to Describing the Functional Requirements of an Embedded Communication System

CONSTANCE L. HEITMEYER AND JOHN D. MCLEAN

Computer Science and Systems Branch
Information Technology Division

June 14, 1982

NAVAL RESEARCH LABORATORY
Washington, D.C.

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NRL Report 8604 | 2. GOVT ACCESSION NO. AD-A116	3. RECIPIENT'S CATALOG NUMBER 738
4. TITLE (and Subtitle) AN APPROACH TO DESCRIBING THE FUNCTIONAL REQUIREMENTS OF AN EMBEDDED COMMUNICATION SYSTEM	5. TYPE OF REPORT & PERIOD COVERED Final report on an NRL Problem 6. PERFORMING ORG. REPORT NUMBER 7590-108:CH:JM:rmr
7. AUTHOR(s) Constance L. Heitmeyer and John D. McLean	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Research Laboratory Washington, DC 20375	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 11402N; X1083SB; NRL Problem 75-0210-0-1
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Electronic Systems Command Washington, DC 20360	12. REPORT DATE June 14, 1982 13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Abstract specification
Communication system
Embedded system
Formal specifications
Functional requirements
Requirements
Requirements document

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An "abstract" requirements specification describes a system's externally visible behavior without making decisions about its design. Although they have important advantages, such specifications are difficult to produce for complex systems and hence are seldom seen in the "real" programming world. This report describes an abstract requirements specification for a complex, real-world system; the specification is intended to serve as a fully worked out example for those tasked to document the requirements of similar systems. After introducing the Navy application with which we are concerned, we demonstrate that the traditional approach of "functional decomposition," where -
(Continued)

20. ABSTRACT *(Continued)*

each output is expressed as a mathematical function of inputs, leads to premature design decisions. Next, we present a new approach to writing requirements documents that avoids design decisions and thus leads to an abstract specification. The new approach is compared to a similar approach used in a related project. An appendix provides an example that illustrates the formal techniques employed in the approach.

CONTENTS

AN APPROACH TO DESCRIBING THE FUNCTIONAL REQUIREMENTS OF AN EMBEDDED COMMUNICATION SYSTEM

INTRODUCTION

Most requirements documents for computer systems say too little about what to do and too much about how do it. Their bulk consists of decisions that should have been postponed until design time instead of precise information about the system's external behavior. Designers are usually forced to devote significant time to clarifying and supplementing the requirements, often making decisions that they are not qualified to make. Moreover, they are either constrained by poor design decisions that have been built into the requirements or forced to glean essential system features from a mass of extraneous detail. The result of this process is expensive, badly designed software that fails to meet the needs of its intended users.

In our view, a requirements document should describe only those demands that the future system must satisfy. Its primary concern should be the externally visible behavior of a computer system; it should not make premature decisions about the system's design. Such a document is useful to software developers because it permits them to choose the most effective implementation. It is useful to maintainers because it distinguishes those features of the system that are dictated by the requirements from those that software personnel are free to change [1].

A requirements document must also satisfy other objectives. It should be clear, precise, easy to modify, and easy to check for completeness and consistency. Moreover, the document must describe any constraints imposed on the implementation: for example, the use of specific computer hardware may be required. Finally, the requirements documentation must be developed in a cost-effective manner; we should not spend more money developing a good specification than we will save by having such a document.

In Refs. 2 and 3 the term *abstract specification* refers to specifications that describe a system's modules without making implementation decisions (for example, decisions about particular algorithms and data structures). In this report we use the term *abstract specification* to refer to requirements documents that describe only what a system is supposed to do, not how it is designed or how it is implemented. Thus an abstract requirements specification excludes decisions such as how the system is divided into modules, what particular data structures are used, and sequencing decisions.

Despite their advantages, abstract requirements specifications may not be practical. Few "real-world" systems possess the simplicity that renders abstract specification straightforward. In many cases, the relation between input and output is so complex that the easiest way to describe it is to suggest a specific method for generating the output from the input, thus forfeiting the advantages of abstraction.

This report presents the results of a project that applied the methodology of abstract specification to a real-world system. The project has two objectives: to determine the suitability of the methodology for applications in which the relation between input and output is complex, and to produce a requirements document as a fully worked out model of an abstract specification.

Manuscript submitted March 30, 1982.

In this project, we took two different approaches to describing requirements. Our initial approach, which we call *functional decomposition*, required that each system output be expressed as a mathematical function of inputs. Because this approach forced us to make a number of decisions about the software design, we abandoned it. As an alternative, we formulated a new approach that describes only what the system does, not how it does it. Thus the new approach leads to an abstract specification.

To date, we have completed a requirements document [4] based on the new approach. Because our system is only one of a class of systems to which the approach can be applied, Ref. 4 serves as a useful model of an abstract specification.

In this report we introduce the application, identify the problems we confronted in specifying its requirements, describe our approach to producing an abstract specification, and compare this approach to a similar one used in the A-7 project [1, 5]. In an appendix, we provide an example that illustrates the formal techniques included in the new approach.

OVERVIEW OF THE APPLICATION

The Submarine Communications Package (SCP) is a small communications system to be installed aboard U.S. Navy submarines. The SCP performs several transformations on encoded message data, converting it to its original, human-readable form and then translating it to the character code required for output. The package will consist largely of computer software, although some parts may be implemented in hardware. It will be embedded [6] in a larger computer system that may differ from one submarine to the next. Thus, different implementations of SCP may be required for different classes of submarines.

Before being received by SCP, message data go through several transformations. They are first converted from human-readable form to encrypted form and are then forwarded to a transmission facility. The facility performs parity encoding, supplies channel information, multiplexes several channels of message data into a single bit-stream, and transmits the multiplexed data over a broadcast link. During transmission, several bit errors may be introduced. The message data are received in the radio rooms of submarines tuned to the appropriate frequency. The SCP is part of the communications system that processes the received data.

The multiplexed message data described above comprise one input to SCP. The radio-room operator provides other inputs, called *configuration data*, that indicate how many channels of message data have been multiplexed, what cryptographic system is associated with each channel, and which channels of data may be discarded. The significance of the configuration data items is that they, along with data that SCP computes internally, determine what transformations SCP should apply to the multiplexed message data to produce the required output.

A REJECTED APPROACH: FUNCTIONAL DECOMPOSITION

Our initial approach to describing the SCP requirements was a traditional one: we generated a functional decomposition. Since the external behavior of the system is too complex to describe with a simple mathematical expression, we needed to produce a mathematical description of each function performed by SCP and to indicate the order in which the functions are invoked. The result is the required mathematical expression of the output in terms of the input. We succeeded in identifying a series of functions that SCP could apply to its input to produce the required output, but we encountered a problem in trying to specify the order in which these functions should be invoked.

To understand the problem, consider two possible solutions. One solution assumes that the functions are invoked in a particular sequence. Its disadvantage is that assuming a particular sequence is an overspecification of the requirements. For example, two functions required of SCP, demultiplexing and channel identification, can be invoked in either order and still produce the same results. Hence, the order in which the functions are performed is a design decision, not a requirement, and as such should not be included in the requirements document. An alternative solution, one consistent with the methodology of abstract specification, is to leave unspecified the order in which the functions are invoked. However, the mathematical specification of a function depends on the format of the input to that function. For example, although demultiplexing and channel identification can occur in either order, the specification of channel identification varies according to whether the identification is done on multiplexed or demultiplexed data. Hence, we could not give a mathematical specification of SCP's individual functions without arbitrarily selecting an order in which the functions are to be invoked.

Another more serious problem became apparent. The functional decomposition that we derived was not unique; other decompositions are possible. Including a single functional decomposition in the requirements document would force us to state that the designer did not have to break the system into the same functions used to specify it. The danger of this approach is that a designer who initially understands the system in terms of the given decomposition might find it difficult to design the system in terms of another decomposition, even if a better one exists. Hence, the requirements document was once again usurping the designer's job.

Both of these problems are exacerbated by the fact that there are constraints on some of the functions; for example, the use of certain cryptographic hardware is required. Thus the designer could ignore neither the functional decomposition nor the order of function invocation offered in the document. The designer would have to be constantly convinced that changes made in the given decomposition did not affect those parts of the decomposition (that is, the constraints) that were not candidates for modification. This could strongly discourage designers from trying functional decompositions different from the original.

USING EXTERNAL FUNCTIONS

Having convinced ourselves of the inadequacy of description via functional decomposition, we abandoned it. Nevertheless, we still needed to describe what SCP was required to do or, more specifically, how to specify acceptable output values.

To accomplish this, we exploited the fact that systems external to SCP perform a series of transformations on the original message data to produce the SCP input. We refer to these transformations as the *external functions.* We decided to describe the SCP output, not in reference to the input (to which it bears no natural relation), but as a function of the original, human-readable message data. Moreover, we also described the input as a function of the original message data. Our approach provides a formal description of each external function and specifies the order in which the external functions are invoked. Describing this order is not an overspecification, because it is fixed by what actually happens externally.

Once the external functions and the order in which they are invoked are described, the formal requirements of SCP are (1) to reconstruct the original data and (2) to convert it to the output character set. The latter step is performed by the *output functions.* By specifying only the external functions and the output functions, the requirements document gives a clear, precise description of the external behavior of SCP, without describing how the system performs the reconstruction. The actual decomposition of SCP into functions and the selection of the order in which functions are invoked are decisions that are left for the software designers.

As an example, consider a system S similar to SCP but with only two external functions. Call the functions f and g, and assume that f is performed first. Then, let $F = g \cdot f$, where $\cdot$ represents functional composition. The requirement of S is that it perform F', where F' is defined to be the inverse of

F, i.e., $F' \cdot F = I$, where I is the identity function. In describing S in this manner, we make no decisions about which function is inverted first. We also leave open the nature of the functional decomposition used to implement F'.

Our original approach, based on functional decomposition, gave us specifications of the form

B implies output := G (inputs),

where B is a predicate and each SCP output is expressed as a function G of one or more input values. The problem was that the complexity of G forced us to make premature design decisions when we tried to specify it as a mathematical expression. In contrast, the new approach, in the simplest case, produces specifications of the form

$$B \text{ implies (input} = F\text{(data) \& output} := \text{data)}, \qquad (1)$$

where F is a function of the original message data. This maintains the discipline of abstract specification while allowing for functional complexity. This form of specification moves the decomposition from SCP to F, which is external to SCP.

In specifying SCP, we actually use a more general form of (1), i.e.,

B implies (input = F (data) & output := H (data)),

where F represents the series of transformations performed by the external functions and H the transformations performed by the output functions. The exact definitions of H and F depend on the condition B that is true. Note that direct specification of H, the translation of the original data to the required output format, does not violate abstract specification. Because of its simplicity, H can be described without decomposing it into smaller functions.

This new approach can be used to give abstract specifications for any one of a class of systems in which there is no simple relation between the input and the output but there is a simple relationship (1) between some set of data D and the input and (2) between D and the output. This includes cases where the input has been encoded, garbled, or otherwise changed.

FURTHER FEATURES OF OUR APPROACH

The generation of a requirements document for SCP led us to consider two broad issues. One issue concerned the general approach to be used; as we have said previously, we needed an approach consistent with the methodology of abstract specification. Another issue concerned what specific formal techniques we should adopt to document the requirements. In the appendix, we describe a sample system that illustrates many of the formal techniques that we formulated and applied in Ref. 4. Here, we briefly discuss some of these techniques. To illustrate two of them, we draw upon parts of the specification given in the appendix.

Table Format

We decided to specify each external function, as well as each output function, in a table. The table format, which is identical for both the external and the output functions, makes it easy to answer specific questions. In addition, it helps in identifying missing information and inconsistencies.

Table 1 is used to specify an external function named MUX. The table consists of several sections. The *syntax* section defines the data types of the function's parameters. The *uses* part identifies any functions that are called by the function being defined, thus making the effects of changes apparent. The *description* section describes in prose what the function does; it motivates the formal

Table 1 — Completed Table for an External Function

External function: Multiplexor

Function short name: MUX

Syntax: MUX(%characterstring%, %characterstring%) --+ %characterstring%

Description: MUX combines two character strings, where each string is associated with a different channel, into a single string. The multiplexing is done on a character basis; that is, in the output, character 1 of the first string is followed by character 1 of the second string, character 2 of the first string, character 2 of the second string, etc.

Uses: None

Notation: $a_1 = a_{1,1} \ldots a_{1,m}$ where $a_{1,i}$ is a %character%
$a_2 = a_{2,1} \ldots a_{2,m}$ where $a_{2,i}$ is a %character%
$b = b_1 \ldots b_h$ where b_i is a %character% and $h = 2*m$

Semantics: MUX(a_1, a_2) = b ==+

(i) (j) (E g)(1 le i le 2 & 1 le j le m ==+

$b_g = a_{i,j}$ & g = (j-1)*2 + i)

Input parameter restrictions: None

description given in the *semantics* section. The specification language used in the semantics section is an extension of first-order predicate calculus. (See the appendix for an explanation of the special notation used in the semantics section.)

Constraints

In addition to specifying what SCP must do, the requirements document also describes the constraints on the implementation. We found that the same table format used to describe the external and the output functions could be used to describe some of the constraints, such as the way in which error detection and correction is implemented.

Not all of the SCP constraints can be described via the tabular format. Some functions required of SCP, for example decryption, must be implemented via special hardware. Thus, Ref. 4 states explicitly that the SCP software must pass data to, and receive results from, special hardware. English prose, rather than formal techniques, is used to describe the constraints on decryption and other similar constraints.

Data Types and Data Items

One section of Ref. 4 defines many different data types. All data objects that exhibit the same externally visible behavior are said to belong to a given *data type*. Reference 4 defines each data type by describing the operations with which it is associated and uses the notation %data type% to identify them. Examples of SCP data types are %characterstring% and %qualityindicator%.

Another section of Ref. 4 defines several data items. Each data item belongs to one of three classes: *input data items*, data items transmitted to SCP from external software; *output data items*, data items transmitted by SCP to external software; and *pre-input data items*, data items that are inputs to one or more of the external functions. These are denoted as /input data item/, //output data item//, and /pre-input data item//.

The SCP requirements document uses a table to describe each data item. The table provides an acronym for the item, identifies its data type, and describes the item in English prose. Table 2 describes an input data item named /MUXDATA/.

Table 2 — Completed Table for an Input Data Item

Input Data Item:	Multiplexed Data
Acronym:	/MUXDATA/
Description:	/MUXDATA/ contains the multiplexed channels of message data. It is the input to T.
Type:	%characterstring%
Error Conditions:	None

"Virtual" External Functions

In some cases, the formal description of the external functions is simplified if we make assumptions that differ from what actually occurs. Thus some of the external functions are virtual.

An example is the external function GARBLE, which is an idealization of the process that introduces bit errors into the message data. The assumption made relative to GARBLE is that no more than a single bit error can occur in each transmitted character. Such an assumption is necessary because of the limited error-correcting capability of the error-correction scheme that SCP is constrained to implement. To avoid confusion, the number of external functions whose description differs from reality was kept small. Moreover, in an early section of the requirements document, we explicitly state each simplifying assumption, along with a description of what actually happens and a list of the external functions that are affected.

COMPARISON WITH THE APPROACH USED IN THE A-7 PROJECT

The SCP requirements document uses many of the techniques that were used in the A-7 requirements document [1, 5]. These include symbolic names for data items and values, special brackets to indicate data items, standard forms, and special tables for consistency and completeness checking. While many of the more general techniques used in Ref. 5 were directly applicable to our document, the technique used to describe the required A-7 functions was not.

In describing functions, the A-7 team faced problems similar to those confronted in the SCP project. The relationship between the A-7 outputs and inputs could be highly complex. Moreover, the A-7 team needed an approach that described the required behavior without making premature design decisions.

The A-7 project team initially planned, as we did, to express each output as a mathematical function of input values. However, they rejected this approach because it forced developers to describe an implementation [1]. We can illustrate this by considering the A-7 function that displays the current altitude. Altitude can be measured in a number of ways. Under some circumstances the A-7 software may read the barometric altimeter to compute altitude, whereas under different circumstances it may be better to read the altitude as measured by radar. The choice is an implementation decision and as such should not be specified in the requirements document.

The technique ultimately used in the A-7 project expresses output values as functions of conditions and events external to the system. *Conditions* are predicates that characterize some aspect of the system for a measurable period of time, but they are not necessarily system inputs; an *event* occurs when the value of a condition changes. In terms of the notation we used earlier, each A-7 function is described via one or both of the following two forms:

$$B \text{ implies output} := k,$$

where k is a constant, or

$$B \text{ implies output} := F(x_1, x_2, \ldots, x_n),$$

where F is a function, and each x_i is either another A-7 output or an externally visible factor of the aircraft or its environment.

Examples of externally visible factors are altitude, air pressure, longitude, ground range, and wind velocity. Some of these factors, for example altitude, can be measured directly and are thus inputs to the A-7 software. Other factors, such as air pressure and wind velocity, cannot be measured directly and must be computed from inputs. A dictionary included in Ref. 5 defines each externally visible factor informally with English prose. For example, longitude is defined as the "longitude coordinate of the present position," and ground range as the "horizontal distance to some point." How to compute these factors is not specificed in the document but is left for the system implementors.

Typically, the function F is quite simple and has only a single parameter. Often, F is the identity function. In other cases, English prose is used to define the relationship between the external factor(s) and the required output. For example, one A-7 output item is defined as "half of !steering error! from Flight Path Marker," where !steering error! is an externally visible factor.

The significant difference between the two approaches lies in the degree to which formalism is used. As noted above, the functional requirements of SCP are defined rigorously in a specification language based on first-order predicate calculus. The definition of the A-7 functions is more informal, relying on the extensive use of English prose and tables to describe the many externally visible factors and the functions themselves.

The more formal techniques used in the SCP document are possible due to the presence of the external functions. Because there is nothing in the A-7 environment that is analogous to the external functions, there is no obvious way to define the A-7 functions more formally.

Likewise, the A-7 approach is not feasible for SCP. In the SCP environment, there are no externally visible factors with a commonly accepted definition; thus Ref. 4 cannot appeal to well-known physical phenomena such as latitude, longitude, and altitude. Instead, it refers to phenomena, such as cryptographic devices and error detection and correction algorithms, whose specific features vary from one application to the next.

While they exhibit some differences, the two approaches also have important similarities. Each provides a clear and precise description of a system's required external behavior. Each leads to a requirements document that is organized for ease of change. Both avoid overspecification: decisions about software design and implementation are not made in the requirements document but are postponed until the later stages of development.

CONCLUSIONS

Even though it seemed impossible initially, a closer look at the SCP application indicated that an abstract specification of the required functions was feasible. The abstract approach produced a precise statement of the requirements and, at the same time, encouraged us to avoid premature design decisions. More generally, the generation of a formal requirements document forced us to seek clarification and further information about the requirements. The result is a document that is much closer to a complete and consistent statement of the SCP requirements than it would have been otherwise.

Our experience also emphasized the importance of having models. As we have stated earlier, many of the techniques used in the A-7 requirements document were directly applicable to our project. The examples provided by Refs. 1 and 5 facilitated our understanding of the techniques and simplified our application of them to a different system.

We hope that, in a similar fashion, Ref. 4 will also serve as a model. There are numerous systems whose function is to undo, in some sense, transformations that have been performed externally. This class includes not only many systems in which input has been multiplexed, encoded, or garbled, but any system that is required to convert data back to some original form. An example of such a system is one that retrieves data that have been hashed into an array. It is more natural to describe such a system in terms of the hashing algorithm used than in terms of a particular dehashing algorithm. Further, such an approach permits an implementation where dehashing is abandoned in favor of, say, linear search of the array. By the application of techniques similar to those illustrated in Ref. 4, it should be possible to produce abstract specifications for the requirements of such systems.

ACKNOWLEDGMENTS

The authors are particularly grateful to K. Britton, formerly of NRL, who encouraged us to write this report and whose comments on earlier drafts led to significant improvements in its readability and structure. We also wish to thank M. Cornwell of NRL, who originally proposed the use of the external functions, and our other NRL colleagues, H. Elovitz, C. Landwehr, D. Parnas, and S. Wilson, for reviewing an earlier version and making many helpful suggestions.

REFERENCES

1. K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Trans. Software Eng.* **SE-6**, 2-13 (Jan. 1980).

2. J.D. McLean, "A Formal Foundation for Trace Specification," *J. ACM*, in publication.

3. D.L. Parnas, "The Use of Precise Specifications in the Development of Software," *Information Processing 77*, B. Gilchrist, ed., North-Holland, New York, 1977.

4. C. Heitmeyer et al., "Requirements of the Submarine Communications Package," NRL Memorandum Report, in progress.

5. K.L. Heninger, J.W. Kallander, J.E. Shore, and D.L. Parnas, "Software Requirements for the A-7E Aircraft," NRL Memorandum Report 3876, Nov. 1978.

6. D.L. Parnas, "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems," NRL Report 8047, June 1977.

Appendix

THE SCP SPECIFICATION TECHNIQUES

INTRODUCTION

This appendix illustrates the specification techniques used in the SCP requirements document [4]. Summarized here is a sample system that performs a small subset of the functions that SCP will perform. The SCP techniques are used to provide an abstract specification of the system's requirements.

The Sample System

The system, named *T*, receives two channels of data that have been encrypted and then multiplexed. The system *T* is .equired to convert each channel of data to its form prior to encryption and then translate the data to the required output format. Using the SCP approach, we must therefore specify two external functions, encryption and multiplexing, and one output function, translation.

Comments on the Specification

To reduce its complexity, the sample system given here describes the needed data types informally. Moreover, we have not included a description of the constraint that decryption be performed by a specific hardware device. The reader is referred to Ref. 4 for an illustration of more formal techniques for specifying such requirements. Finally, we have not included a specification for the function CTABLE, which defines the mapping between the original character code and the output character code.

T: REQUIREMENTS SPECIFICATION

The specification is divided into four sections. Section 1 identifies the data types and data items that T requires. Section 2 specifies two external functions, ENCRYPT and MUX, while Section 3 describes the output function, TRANSLATE. Each function is specified via a table; the notation used in the semantics field of each table is defined in Table A1. Section 4 defines the relationship between T's input and its output using the external functions, the output function, and the data items.

Table A1 — Notation Used in Function Tables

Symbol	Meaning
(i)	for all i
(E g)	there exists g
==+	implies
--+	is mapped to
le	is less than or equal to
&	logical AND

1. Data Types and Data Items

The only data types used in the specification are the standard ones, %character% and %characterstring%. The data items required to specify T are presented below.

1.1 Pre-Input Data Items

1.1.1

Pre-Input Data Item: Data for Channel i

Acronym: /DATAi// i = 1,2

Description: /DATAi// is the original, human-readable data, associated with channel i, before it has been transformed by any of the external functions.

Type: %characterstring%

Error Conditions: None

1.1.2

Pre-Input Data Item: Encrypted Data for Channel i

Acronym: /ENCDATAi// i = 1,2

Description: /ENCDATAi// is the data associated with channel i after it has been encrypted and before it has been multiplexed.

Type: %characterstring%

Error Conditions: None

1.2 Input Data Item

1.2.1

Input Data Item: Multiplexed Data

Acronym: /MUXDATA/

Description: /MUXDATA/ contains the multiplexed channels of message data. It is the input to T.

Type: %characterstring%

Error Conditions: None

1.3 Output Data Item

1.3.1

Output Data Item: Processed Data for Channel i

Acronym: //PROCDATAi// i = 1,2

Description: //PROCDATAi// is the original, human-readable data associated with channel i after it has been translated to the output character set. It is the output of T.

Type: %characterstring%

Error Conditions: None

2. External Functions

2.1

External function: Encryptor

Function short name: ENCRYPT

Syntax: ENCRYPT(%characterstring%) --+ %characterstring%

Description: ENCRYPT is an encryption function performed on a character-by-character basis by the XYZ encryption device. ENCRYPT maps each character of the input string into a character in the same relative position in the output string. The mapping is performed in such a way that the XYZ decryption device can perform the inverse operation.

Uses: None

Notation: $x = x_1 ... x_n$ where x_i is a %character%
$y = y_1 ... y_n$ where y_i is a %character%

Semantics: ENCRYPT(x) = y ==+

(i) (1 le i le n ==+ y_i is the XYZ-encryption of x_i)

Input parameter restrictions: 1 le n le 5000

2.2

External function: Multiplexor

Function short name: MUX

Syntax: MUX(%characterstring%, %characterstring%) --+ %characterstring%

Description: MUX combines two character strings, where each string is associated with a different channel, into a single string. The multiplexing is done on a character basis; that is, in the output, character 1 of the first string is followed by character 1 of the second string, character 2 of the first string, character 2 of the second string, etc.

Uses: None

Notation: $a_1 = a_{1,1} \ldots a_{1,m}$ where $a_{1,i}$ is a %character%
$a_2 = a_{2,1} \ldots a_{2,m}$ where $a_{2,i}$ is a %character%
$b = b_1 \ldots b_h$ where b_i is a %character% and h = 2*m

Semantics: MUX(a_1, a_2) = b ==+

(i) (j) (E g) (1 le i le 2 & 1 le j le m ==+

$b_g = a_{i,j}$ & g = (j-1)*2 + i)

Input parameter restrictions: None

3. Output Function

3.1

Output function: Translate

Short name: TRANS

Syntax: TRANS(%characterstring%) --+ %characterstring%

Description: TRANS translates a character string from one character set to another. Each character in the input string is replaced by the appropriate character to produce the output string.

Uses: CTABLE

Notation: $x = x_1 \ldots x_n$ where x_i is a %character%
$y = y_1 \ldots y_n$ where y_i is a %character%

Semantics: TRANS(x) = y ==+

(i) (1 le i le n ==+ y_i = CTABLE(x_i))

Input parameter restrictions: None

4. Specification of T

4.1 Specification of T's Input

(1) For i = 1,2

/ENCDATAi// = ENCRYPT(/DATAi//)

(2) /MUXDATA/ = MUX(/ENCDATA1//, /ENCDATA2//)

4.2 Specification of T's Output

For i = 1,2

//PROCHANi// = TRANS(/DATAi//)